Repo:
An Interpreted Language for Exploratory Programming of Highly Interactive, Distributed Applications

Blair MacIntyre
GVU Center, College of Computing, Georgia Institute of Technology
blair@cc.gatech.edu

Abstract

In this paper we present Repo, an interpreted language for exploratory programming of distributed interactive applications. Repo is based on Obliq, a distributed language that supports client-server distribution semantics of all data items (objects, arrays and variables). Repo extends Obliq's type system uniformly so that all its data items can also be distributed with unsynchronized or synchronized replication semantics, both of which are needed by highly interactive applications. Since all of Repo's data items can take on any distribution semantic and be mixed in arbitrary ways, a wide range of interesting data structures can be developed in Repo in a straightforward manner. Since Repo allows distributed applications to be developed in a few lines of interpreted code, it turns out to be an excellent language for exploratory programming of distributed interactive applications. We discuss the design and implementation of Repo, and provide illustrative examples taken from prototypes build using it.

1 Introduction

In recent years, the ever-decreasing cost of computers has allowed researchers to begin exploring a new class of user-interaction paradigms inspired by Weiser's notion of *ubiquitous computing* [30]. Proponents of ubiquitous computing envision a future where computers are cheap, plentiful and can be used together effortlessly. Unfortunately, while hardware is finally small and cheap enough to begin exploring such problems, the corresponding software tools have not matured at the same rate. In particular, languages and tools for doing exploratory programming of interactive applications that coordinate input and output across a potentially large number of displays (and their corresponding machines) are spare, especially compared to the tools available for single process development.

In this paper we present Repo, a distributed, interpreted language that we developed for prototyping distributed, highly interactive applications such as these. An important requirement of such applications is their need for replicated data, which Repo satisfies by allowing all data objects to be transparently replicated. Repo is based on a language called Obliq [5], a lexically scoped distributed language that supports transparent client-server distribution semantics for all data items (objects, arrays and variables). Repo enhances Obliq's type system by allowing all data items to have one of three distribution semantics: client-server, strictly synchronized replication and unsynchronized replication. Since a data item's distribution semantics are transparent, data items with different distribution semantics can be mixed and matched in arbitrary ways, allowing interesting distributed data structures to be created in a small amount of code. Repo also includes a number of libraries that are needed to support rapid prototyping in our domain, such as simple support for reflection, HTTP clients and servers, regular expressions and so on, most of which we will not discuss here.

In the remainder of this paper, we will describe Repo, often by contrasting it with Obliq. While we will provide enough information about Obliq that the reader can appreciate Repo's design, there are many aspects to Obliq that are not changed in Repo, and will therefore not be discussed in depth. For a more in depth discussion of Obliq, and examples of it in use, see [5]. First, in Section 2 we will provide background describing the specific problem Repo was designed to address, and the system (called Coterie) in which it is integrated. In Section 4 we will discuss other distributed languages, both interpreted and compiled. We will then turn our attention to the design of Repo, focusing on how it cleanly extends Obliq to support replicated data. An overview of Obliq and Repo will be presented in Section 5, followed in Section 6 by a discussion of how support for replication in Repo changes the distributed semantics of Obliq. In Sections 7 and 8, we will briefly describe the changes to the Obliq object declarations and some addi-

tional features necessary for replications (such as custom object picklers and change notification). In Section 9, we will present a number of illustrative examples of Repo in use. Finally, in Sections 10 we will close the paper with a discussion of our experiences with Repo and our plans for future work.

2 Background and Motivation

Our research group[1] has been building on our previous work in *augmented reality* [e.g., 8, 9] by exploring how see-through, head-worn displays can be integrated with the notion of ubiquitous computing to form multi-user *augmented environments*. Such environments combine traditional desktop computer displays with a variety of hand-held, wall-sized and see-through head-worn displays to form a cohesive interaction space in which information is presented to the users on combinations of these displays [22]. Unfortunately, integrating multiple users, multiple displays of different kinds (including head-worn displays that require the user's head to be tracked), and a wide variety of input devices (from pens and mice to voice to three and six degree-of-freedom (DOF) trackers) into a single cohesive system can be extremely challenging.

Even if the technical details of dealing with such a wide variety of devices are ignored, the mere fact that they are attached to an assortment of computers implies that even the simplest of applications must be distributed across these machines. In addition, the highly interactive nature of these applications requires that they support replicated data; since the graphics on many of the displays (especially the see-through head-worn displays worn by some users) must be updated as fast as possible (many times per second) in response to changes in the environment (such as a user's head motion), all of the data used to update the display must be located in the process performing the update. Since much of this data is needed by more than one process, it must therefore be replicated across all interested processes.

The complexity of building distributed programs containing replicated data is further exacerbated by the exploratory nature of prototyping applications for a completely new interaction paradigm; neither the structure of the applications, the kind of data being shared, nor the distribution characteristics of that data are necessarily known ahead of time and will likely be modified frequently as the applications are developed. While building any interactive application typically requires the program to undergo numerous iterations through the build-deploy-evaluate-redesign cycle, building research prototypes often requires that these redesigns take the application in radically new directions. Furthermore, to adequately explore an interaction space, many different approaches should be tried, requiring numerous different prototypes to be built. If any change or new prototype is exceedingly difficult to implement, it may not be explored and the research will suffer.

Ideally, minor conceptual changes should only require a minor amount of work on the part of the programmer. When building distributed systems, the most severe example of when minor changes cause significant work is when the distribution characteristics of a data structure are changed (i.e., when a non-shared data structure needs to be shared, or a client-server data structure needs to be replicated). While the code should be impacted as little as possible, many distributed programming tools deal with local, client-server and replicated data in very different ways, requiring significant changes to the code when the semantics change. If we are to avoid having to make substantial changes, the programming environment should exhibit a high degree of *network data transparency,* requiring the programmer to know as little as possible about the distribution characteristics of a piece of data in order to use it.

Repo is part of a development environment, called Coterie, designed to allow distributed interactive systems to be prototyped as easily as non-distributed ones [20]. To accomplish this, Coterie (and Repo) provides simple to use, highly transparent data distribution, even when this means the network usage of those programs may not be as efficient as would otherwise be possible [21]. This focus on ease of use, at the expense of efficiency, is not typical in the design of distributed programming environments. However, as mentioned above, our target audience is HCI researchers, most of whom have little experience (or direct interest in) building complex distributed applications. Such programmers are not overly focused on the efficiency of execution of the prototypes they are implementing as

1. This research was performed while the author was a member of the Computer Graphics and User Interface Lab in the Computer Science Department at Columbia University.

long as they are “fast enough.” However, they are very concerned with the ease of evolving their prototypes to explore new research directions.

Coterie was written in the Modula-3 programming language [12]. The decision to use Modula-3 was based on the language itself and the availability of a set of packages that provided a solid foundation on which to base our research[1]. Modula-3 is a descendent of Pascal that corrects many of its deficiencies. In particular, Modula-3 retains strong type safety, while adding facilities for exception handling, concurrency, object-oriented programming, and automatic garbage collection. One of its most important features for our work is that it gives us uniform access to these facilities across many architectures.

Coterie’s distributed programming model is based on a familiar and well understood non-distributed programming paradigm, that of multiple threads of control communicating via shared objects. By providing an object-based implementation of *distributed shared memory* (DSM) [18], often called a *distributed object memory* (DOM) [17], both stand-alone and distributed programs are built the same way, with local and distributed data being used transparently and interchangeably, and with threads on the same or different machines communicating through shared objects. Coterie’s DOM is provided by a combination of the Network Object (client-server semantics) and Shared Object (synchronized replicated semantics) object distribution packages. Both of these packages support the creation of transparently distributed Modula-3 objects via compile time code generation and a runtime support library. Both also allow simple Modula-3 data structures to be passed between sites, giving us support for unsynchronized replicated data. The Shared Object package was designed by us to address the needs of highly interactive, distributed applications [23]. Repo (as other Coterie packages, such as Repo-3D, a distributed 3D graphics library [24]) is implemented on top of these two packages.

3 Synchronized Data Replication: The Shared Object Package

In this section, we will briefly describe the Shared Object package, as its design directly influences how we expose synchronized replicated data into Repo. We will not justify the design of the Shared Object package here; the interested reader is referred to [23] and [21]. The Shared Object design was inspired by the approach to object replication used by Bal and his colleagues [2]. In their formulation, implemented in the Orca programming language [1], objects are replicated across machines as needed and the semantics of object replication are enforced by the language. Replication consistency is accomplished in both of these systems via a write-update protocol based on function shipping and totally ordered group communication: methods that update an object are applied to all replicas in the same order. Methods that do not change an object are applied only at the site that executed them. In these systems, shared state is encapsulated in objects and that state is accessed through object methods.

As it turns out, this approach is extremely well suited to implementation as an add-on to a strongly-typed programming language such as Modula-3. Furthermore, the performance characteristics of this approach are appropriate for highly interactive graphical systems, where the objects tend to have a high read/write ratio and need low latency update distribution. To create a replicated object, a programmer simply informs the code generator and runtime systems (via code annotations) which methods update the object state and which simply read that state. The code generator creates code that, in conjunction with the Shared Object runtime system, ensures that the methods that update the object are executed at all sites in the same order. Methods that read an object are executed immediately in the site at which they are invoked. The Shared Object runtime also takes care of ensuring that only one copy of an object can exist at any given site, and is optimized for the case when a replica already exists at a site. When a replicated object is passed to a site that already contains a replica of that object, only a small object identifier is sent across the network, and the existing replica is used at the destination site.

By encapsulating application state in the language objects and having the semantics enforced transparently, the Shared Objects satisfy one of our primary goals by exhibiting a high degree of network data transparency. Because these objects are tightly integrated into the programming language, objects with different distribution semantics can be mixed in arbitrary ways with predictable results.

1. Today, we may have chosen Java as our implementation platform, for similar reasons.

4 Related Work

There have been many interpreted procedural languages created over the years, and a number of them have supported, or been extended to support, client-server data distribution. For example, two of the most popular interpreted languages, Tcl [27] and Python [29], have been extended with support for distribution via client-server semantics (Python via ILU [14], and Tcl via custom extensions such as [25] and Tcl-DP [28]). Unlike these language extensions, Obliq was designed from the start for distributed programming. Obliq has an elegant model of computation built around the use of lexical scoping and higher-order functions in a distributed context, as will be explained in Section 5. Unfortunately, Obliq supports only client-server data sharing.

We are interested in interpreted languages that present an object-oriented or procedural programming model, including support for data replication. To our knowledge, no other such languages exist. There have been distributed interpreted languages that present the programmer with programing models that differ from the usual procedural style, especially in the Agents community (e.g., Telescript [33], and Agent Tcl [10]). However, these languages provide support for distributing computation through code mobility, and do not support building complex distributed applications needing efficient replicated data.

There has been much more work on distributed, object-based programming for compiled languages, much of which has focused on client-server semantics. Most of the early distributed languages (e.g., Argus [19] and Emerald [15]) supported only client server distribution of objects. Many distributed programming toolkits have been designed to work with, or are enhanced versions of, existing languages. RPC [3], CORBA [26] and ILU [14], are designed to be language independent, whereas Network Objects [4], Distributed Smalltalk [6], RMI [34] and the Penumbra toolkit [16] are designed to work with a specific language (Modula-3, Smalltalk, Java and C++, respectively). Because they are tightly integrated with a single programming language, these toolkits typically provide the features of that language on a distributed scale, such as distributed garbage collection, exception propagation between sites, support for marshalling of complex arguments, and so on.

While there have been a number of languages created that support replicated data, most systems are implemented as libraries that can be linked with programs written in an existing sequential language. A large number of parallel and distributed languages exist that extend sequential object-oriented languages such as C++ (many are discussed in [32]). Many, such as Mentat [11] and other non-C++ derived languages such as Distributed Oz [13], use a programming model that is not tightly integrated into the object model of the original language. While such a design may support more efficient, fault-tolerant, scalable distribution mechanisms, it destroys the network data transparency we feel is vital to support exploratory programming.

There have been few languages or language extensions that tightly integrate replicated data into the object model of the language. The Shared Object package for Modula-3 (on which Repo is based), and the Orca language on which this package was modelled, support replicated objects in a transparent way.

5 An Overview of Obliq and Repo

Obliq is a lexically-scoped, untyped, interpreted language for distributed object-oriented computation. It is implemented in, and tightly integrated with, Modula-3, the compiled language in which Coterie is built. Obliq uses, and supports, the Modula-3 thread, exception, and garbage-collection facilities. Its distributed-computation mechanism is implemented using Modula-3 Network Objects, allowing transparent support for multiple processes on heterogeneous machines. An Obliq computation may involve multiple threads of control within an address space (process), multiple address spaces on a machine, heterogeneous machines over a local network, and multiple networks over the Internet.

The guiding principle that separates Obliq from other distributed procedural languages is its adherence to lexical scoping in a distributed higher-order context. This principle is conceptually simple and has a number of interesting consequences: it supports a natural and consistent semantics of distributed computation, and it enables elegant techniques for distributed programming. Lexical scoping ensures that the binding location of every identifier can be determined by simple analysis of the program text surrounding the identifier. Therefore, the meaning of program identifiers can be determined when they are introduced, not when they are used, allowing programmers to

reason about the behavior of their programs, even when they are widely distributed and involve many simultaneous threads of control.

It does not matter where an identifier is used, since it always refers to the binding location *and* network site at which it was created. This is especially important when higher-order functions with free identifiers are transmitted over the network. Lexical scoping implies that these free identifiers are bound to variables when the higher-order function is analyzed, not when the function is executed. Therefore, higher-order functions are always self-contained as they move around the network, carrying along references to the variables referenced by their free identifiers.

Both Obliq and Repo support uniform semantics across all data types, including objects, arrays and variables. As we noted in [21], and as Wilson and Bal point out in their evaluation of replicated objects in Orca [31], this ability to share not only objects, but arrays and variables, simplifies many standard programming tasks. Since Repo extends the Obliq data model to include both synchronized and unsynchronized replicated objects, Repo data items have state that may be local to a site (as in Obliq) or replicated across multiple sites. The syntax and semantics of Repo differs as little as possible from Obliq, although the addition of replicated data does involve some conceptual differences. We will discuss the changes to the semantics of Obliq in Section 6, and to the syntax in Section 7.

6 Distributed Semantics

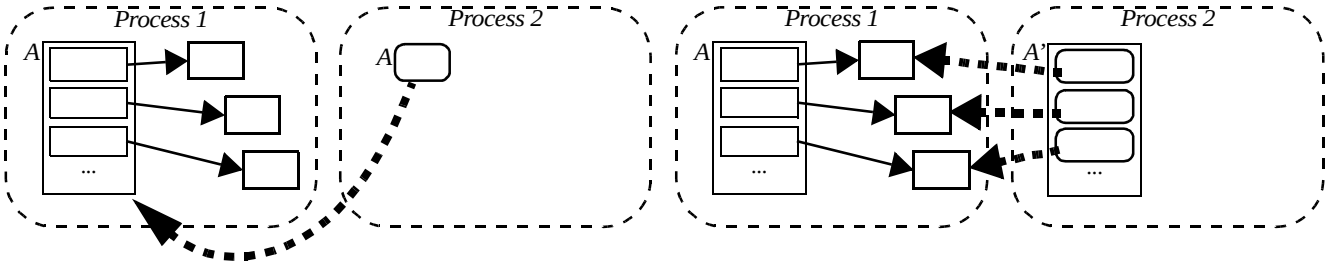
As discussed above, Repo is a descendant of Obliq that extends the Obliq object model to include replicated objects, both synchronized and unsynchronized. In this section we will discuss the distributed semantics of Repo, focusing on how they differ from Obliq as a result of the addition of replicated data. In this discussion, a network address is a pair consisting of a *site address* (the process running on some machine) and a *memory address* at that site. The semantics of Obliq data can be described consistently by considering all addresses to be unique network addresses. Obliq data structures are assembled out of network addresses, just like ordinary data structures are assembled out of local addresses (more precisely, the implementation is designed to create this illusion). As data structures are passed around the network, the embedded network addresses do not change. For example, if an object is passed to another site, the value received at the remote site is a network address referring to the object at the original site. Data items can be explicitly copied between sites (creating new objects at new network addresses), but are never copied implicitly.

The semantics of Repo data are slightly more complicated because of the introduction of replicated data. Repo supports the following three distribution semantics when objects are transmitted from one site to another:

- *remote* objects, whose state exists at one site and are accessed remotely via remote method calls. In Obliq, all objects are remote.
- *replicated* objects, whose state is replicated at all sites that have references to them, with consistency enforced across all sites by ensuring all updates are applied in the same order to all replicas. When transmitted between sites, these objects are implicitly copied and new network addresses are created.
- *simple* objects, whose state is replicated at all sites to which they are transmitted, but do not have consistency enforced across these sites. When transmitted between sites, these objects are implicitly copied and new network addresses are created.

In Repo, we use the term *replicated* to refer to synchronized replicated objects, and the term *simple* to refer to unsynchronized replicated objects. We selected these terms because Obliq already used the term *synchronized* to refer to objects with an implicit mutex around all method calls. The term *simple* arises from the fact that these objects correspond to the simplest of all possible distribution semantics, in which data is copied between sites with no further action required.

As mentioned above, when Obliq data is transmitted around the network, the network addresses embedded in the data do not change, always referring to the original data item at the original site. Repo objects, however, can have embedded network references to replicated data. When a reference to an unsynchronized replicated data item is transmitted across the network, a new copy of the data referred to, with a new network address, is created at the destination site. Therefore, any embedded network references to this unsynchronized replicated data will be



(a) Object *A* has client-server semantics (b) Object *A* has replicated semantics

Figure 1: The effect of different distribution semantics. When an object *A* is copied from *Process 1* to *Process 2*, the result depends on the distribution semantics. For simplicity, assume all of the embedded references in object *A* are to client-server objects. In (a), *A* is a client-server object, so the network address is copied to *Process 2*, and all access to object *A* refers back to the original object. In (b), *A* is a replicated object, so a new replica is created and the embedded references in *A* are copied recursively. In this case, since the references are to client-server objects, their network addresses are copied and all access refers back to the original objects.

changed to refer to the new local address. If a reference to the same unsynchronized replicated data item is sent to a process multiple times, multiple new, independent replicas will be created.

When a reference to a synchronized replicated data item is transmitted across the network, the system first checks to see if a replica of this object exists in the destination site. If a replica exists, its network address is substituted for any embedded references to this data object. If a replica does not exists, a new replica, with a new network address, is created and substituted for any embedded references to this data object. All replicas of a synchronized object maintain an association with each other, even though they have different network addresses.

Consider the following example, to help clarify the differences in the semantics, illustrated in Figure 1. Assume we have an array that we wish to distribute to a number of processes. If the array has client-server semantics, when references to it are passed around the network, only its network address is distributed, and all access is to the original array. If the array is replicated, when references to it are passed around the network, it is replicated. The process of replication causes its elements to be sent to the new site, which causes the process to be repeated recursively: if an element is a client-server data value, only its network reference is sent to the new site, but if the element is a simple constant or a replicated data value, it is copied to the new site, with its elements in turn copied recursively, and so on. Since arrays and objects with different semantics can be mixed arbitrarily, interesting and powerful data structures can be built in a few lines of code.

The different distribution semantics also manifest themselves to the programmer by weakening the guarantee of correct execution that Obliq provides: in Obliq, computations are guaranteed to give the same result no matter where they are executed on the network[1]. Obliq can provide this guarantee because of the use of client-server data and lexical scoping: when program code is evaluated (either within object methods or procedures), its free variables are bound to data items and the network addresses of those data items are embedded in the function closure (the data structure representing the evaluated code). As the closure is passed around the network, it carries these network references with it, and they refer to the same data objects no matter where the closure is executed. Therefore, evaluating this closure always gives the same results, independent of the execution site.

The introduction of unsynchronized replicated data weakens this guarantee. If a function closure is sent to a remote site for execution, and some of its free variables are bound to unsynchronized data, those data values will be replicated at the remote site and the new network addresses substituted for the old ones in the closure. If the function does not modify the data object, the correct execution guarantee holds. However, if the function modifies one of these data items, the replicas at the original site will not reflect these changes, resulting in program execution that differs depending on the execution site (since data at different network addresses is being modified).

1. While this guarantee is useful, it is a simplistic one, since it necessarily assumes no built-in libraries are accessed that give different results at different sites. For example, if a computation accesses the file system, it may not find the same files at different sites.

objects:	{x_1 => a_1, ... ,x_n => a_n} every field of an object has state access: a.x, a.x(a_1, ... ,a_n) update: a.x := b, delegate a to b end
arrays:	[a_1, ... , a_n] every element of an array has state access: a[n] update: a[n] := b
variables:	var x = a variables have state (identifiers declared by "let" do not) access: x update: x := b

Table 1: Entities with state in Obliq. There are three kinds of entities that have state in Obliq: objects, arrays and variables. These entities are declared, accessed and updated as shown. The `delegate` update syntax redirects the fields of `a` to access the fields of object `b` (in this case), and is used to support a simple form of object migration.

While programmers need to be careful when they use unsynchronized replicated data, the loss of this correctness guarantee is largely a pedagogical one; this guarantee is primarily a useful way of explaining and understanding how lexical scoping affects program behavior. We will return to this point in Section 10. The primary reason unsynchronized replicated data is provided is for efficient access to immutable data objects, which (by definition) will not be modified. We have also found other uses for unsynchronized data, some of which will be shown in Section 9.

7 Replication Syntax

When Repo was original designed, we made a decision to retain as much of the Obliq syntax as possible, with the goal of having all Obliq programs be valid Repo programs. With one small exception (arising from another enhancement unrelated to data replication), we succeeded. In this section we will briefly describe how the introduction of support for replicated data affects two areas of Obliq syntax: declarations, and commands for *cloning* (copying) data.

7.1 Declarations

Repo syntax differs from Obliq syntax primarily in the way data items are declared. In Obliq, there are three kinds of data items that can have state (shown in Table 1), and are thus affected by the addition of support for replication. These declarations are also valid Repo declarations, and create client-server entities.

To allow programmers to select different distribution semantics, we added the `simple` and `replicated` modifiers to these declarations, as shown in Table 2. Since replicated objects are implemented using the Shared Objects package, we also need to decide which actions update these entities (see Section 3). In the case of arrays and variables, the decision is straightforward and intuitively obvious: the access operations shown in Table 1 read from the entities, and the update operations update them. In the case of objects, the decision is slightly more complex.

The access operations for objects shown in Table 1 correspond to reading fields and invoking methods, while the update operation corresponds to changing the value of a field. Like arrays and variables, we define the operations of reading and updating fields as read and update actions, respectively. This differs from the semantics of Shared Objects, where only the update method calls are distributed in this way. However, being able to create simple replicated objects without the need to define methods to update the data fields is convenient, and if the programmer wishes to restrict access to the data fields, they can declare the object as `protected`, which prevents the data fields from being modified from outside the object methods. Alternatively, lexical scoping can be used to

Objects	
client-server:	{x_1 => a_1, ... ,x_n => a_n}
protected:	{protected, x_1 => a_1, ... ,x_n => a_n}
serialized:	{serialized, x_1 => a_1, ... ,x_n => a_n}
synchronized:	{replicated, x_1 => a_1, ... ,x_n => a_n}
unsynchronized:	{simple, x_1 => a_1, ... ,x_n => a_n}
Arrays	
client-server:	[a_1, ... , a_n]
synchronized:	replicated [a_1, ... , a_n]
unsynchronized:	simple [a_1, ... , a_n]
Variables	
client-server:	var x = a
synchronized:	var replicated x = a
unsynchronized:	var simple x = a

Table 2: Declaring entities with state in Repo. Repo has the same three kinds of entities with state as Obliq: objects, arrays and variables. These entities are accessed and updated in the same was as they are in Obliq. By default, these entities have Obliq's client-server distribution semantics. Additional keywords are used to declare synchronized and unsynchronized replication semantics, as shown. As with Obliq, if an object is declared to be *protected*, its data fields can only be changed internally by its own methods and it cannot be cloned. If an object is declared *serialized*, there is an implicit lock around its methods that limits access to one thread at a time. Replicated objects can also be declared as protected and are automatically serialized (using a mutex automatically created by the Shared Object package). Simple objects can be declared as protected and/or serialized.

define object data that is not contained in the object fields, and can therefore not be accessed from outside of the object.

The other access operation on an object is method invocation. As with Shared Objects, methods are the primary means of updating and accessing objects. To differentiate between methods that update an object, and those that do not we added an update method declaration, denoted with the `umeth` keyword. Methods created with the original Obliq method syntax, denoted with the `meth` keyword, are treated as read methods, and those defined using the new `umeth` keyword are update methods, and are therefore applied to all replicas of the object.

7.2 Cloning Data

In Obliq, once an object is declared it cannot have fields added to it, nor can it be moved from the site at which it was created. However, Obliq supports object *cloning*. When an object is cloned, a new object is created with the same field names, and the fields are initialized to refer to the same values (methods, data or aliases) as the original object. Multiple objects can be cloned together to form a single new object, with the restrictions that all of the field names must be unique across the set of objects. Similarly, when arrays are cloned they cannot have their size changed. To change the size of an array, it must have a second array concatenated to it to create a new array containing the elements of both[1]. The new object or array is created at the site where the operation is executed, which need not be the same site as that of the objects or arrays being copied.

The decisions to have objects and arrays be immobile and structurally immutable were made to simplify the implementation and to keep the language clean and predictable. Cloning objects and concatenating arrays result in the creation of new data elements. If the old elements are in use, they will continue to exist unchanged; if they are not longer used, they will eventually be garbage collected.

In Repo, we must define what happens when multiple objects are cloned, or when multiple arrays are concatenated, and they do not all have the same distribution semantics. For example, what happens when we concatenate a remote array (`a1`) to a replicated array (`a2`) (i.e. `a3 := a2 @ a1`)? Or solution, when concatenating arrays, is to

1. A copy of an array that is the same size can be created by concatenating the array to the empty array. New arrays are also created by extracting a subarray of an array, but we will only refer to concatenation for simplicity.

have the new array adopt the semantics of the array to which it is concatenated: in this example, the result (`a3`) is a replicated array. The decision is not so simple with objects, because we need a way of specifying update methods for replicated objects: for example, if we clone a simple object to a replicated object to create a replicated object, we may want some of the fields of the simple object to be considered update methods in the resulting replicated object.

Therefore, we require that all objects have the same semantics if they are to be cloned together, and provide operators to convert an object from one distribution semantic to another. These new operations (`remote(a1)`, `replicated(a1,`*umeth-list*`)`, and `simple(a1)`) do not modify the semantics of an existing objects; rather, they each take their object argument (`a1`) and return a clone of that object with the appropriate distribution semantics (client-server, synchronized replicated or unsynchronized replicated, respectively). In addition, the `replicated` operator takes a second parameter, which is a list of the field names of methods to be converted from methods (created with the `meth` keyword) to update methods (that would have been specified with the `umeth` keyword had this object been originally created as a replicated object). For example, consider the following object:

```
let o1 = {simple,
            data => 1,
            get => meth (s) s.data end,
            set => meth (s, val) s.data := val end};
```

We could create a replicated version of this object as follows:

```
let o2 = replicated (o1, [“set”]);
```

This would give us the same object as this definition:

```
let o2 =
      {replicated,
       data => 1,
       get => meth (s) s.data end,
       set => umeth (s, val) s.data := val end};
```

For convenience, we also allow arrays to be used as arguments to these three conversion operators, in which case all three of the operators take the array to be cloned as their single parameter.

8 Additional Replication Support

To properly support replication in Repo, we needed to support two more features: change notification and object serialization (or *pickling*, as it is known in the Modula-3 community). Change notification was supported as part of a new built-in `replica` module in Repo. Among other things, this module provides functions to create and destroy Repo callback notification objects (`replica_notify` and `replica_cancelNotifier`). The `replica_notify` function takes the replicated object and a simple (unsynchronized replicated) object to use as its notifier. The methods of the notifier object correspond to the updates that the programmer wishes to be informed of, where each method in the original object (for example, `myMethod`) can have corresponding methods in the notifier object that are called just before (`pre‘myMethod`) or just after (`post‘myMethod`) the method in the original object is called. A pair of catch-all methods are also available (`pre‘anyChange` and `post‘anyChange`) that allow programmers to be notified of any updates not handled by a more specific notification method.

In Obliq, copying an object from one site to another is always the result of an intentional action by the programmer (either cloning an object or concatenating an array). Therefore, it is left to the programmer to control what data is copied between processes when they create new objects or arrays in different processes, so no automated support for object pickling is provided. In Repo, on the other hand, replicated data can be copied implicitly when data is passed between machines. Therefore, we need to provide some way for programmers to control what is copied. To support pickling of replicated objects, we define the `objectpickler` command. In Modula-3, pickling is done by two routines, one that writes the object to a byte stream, and one that reads the object from a byte stream.

Rather than write two routines for reading and writing objects from and to byte streams, which would be cumbersome and inefficient in Repo, the programmer creates two simple objects for reading and writing the object. For

```
let rep = {simple,
  x => 0.0, y => 0.0, z => 0.0};

let trackerDist = {replicated,
     data => rep,
     set => umeth (self, val)
          self.data := val;
        end,
     get => meth (self)
          self.val;
        end
  };
```

(a) An implementation using a straightforward pair of access methods and a data field.

```
let trackerDistData = {replicated,
     data => rep
  };

let trackerDistMsg = {replicated,
     set => umeth (self, val) end
  };

let trackerArray =
  replicated [rep];
```

(b) Alternate implementations.

Figure 2: An example of synchronized replicated objects in Repo. A basic implementation, `trackerDist`, is shown in (a), including a prototype tracker report object `rep`. Of course, any valid Repo object could be used for `rep`, but this object is shown for clarity. The `set` method is an update method, while the `get` method is not. Alternate implementations are shown in (b). Since data fields of Repo's replicated objects can be manipulated directly, the object could also be defined with no methods, as in `trackerDistData`. A stateless version of the object, `trackerDistMsg`, allows the data to be distributed via the `set` method without storing it in the object. A version using an array, `trackerArray`, is also shown.

each data field in the original object, these two new objects must have a corresponding method that takes a single parameter and returns a Repo value. When the object is being pickled out to the network, the writer object methods are passed the current value of the corresponding field in the object, and the return value is written to the network. When the object is being pickled in from the network, the reader object methods are passed the value that was read in from the network (the value returned by the corresponding writer object method), and the return value is assigned to the corresponding field in the object.

9 Examples

In this section we will give a number of examples, taken from prototype applications that have been built with Repo, to illustrate both the simplicity and flexibility of Repo's object distribution semantics, and how the three distribution semantics can be linked together in straightforward and powerful ways.

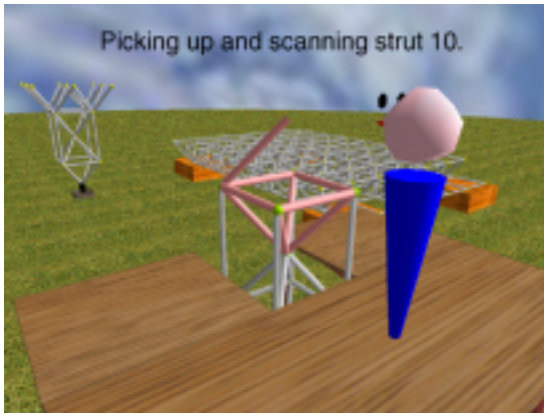
9.1 Simple Tracker Report Distribution

A common problem is distributed augmented environments is how to efficiently distribute the reports from various tracking devices to the processes that are interested in them, and to only those processes. In Repo, this task is straightforward: create a replicated data item to contain the tracker data, and any process that receives a copy of that object will receive the tracker reports. When a local replica of the object is garbage collected from a process, that process stops receiving the tracker reports.

Three simple objects that accomplish this task in slightly different ways are shown in Figure 2. `trackerDist` is the obvious implementation of such an object, with a `data` field containing the current tracker report value, and two methods to access that data field. However, for a simple object such as this, the access methods are not needed. `trackerDistData` is the same object with the methods removed, which would have the programmer access the data fields directly. Of course, we would probably want to use the former version, so that more complex objects (such as that support local filtering or interpolation of tracker reports) could be substituted without changing other parts of the code.

A final variation of the `trackerDist` object highlights an interesting feature of the replication model. Occasionally, when an object such as this is created, the programmer only cares about changes to the data field, but never accesses the data directly. In this case, we can create an object with a `set` method that does nothing with its

(a) What the worker sees through their head-worn display during the installation of strut 11.



(b) What a remote expert sees on their desktop display when the worker is installing strut 10.

Figure 3: A space frame prototype for remote consultation.

argument. When this method is invoked, the arguments are marshalled and distributed to all processes containing replicas of the object. When the method is invoked on the replicas, any callback notification objects have their appropriate methods invoked as well, as discussed in Section 8. In effect, one can view such objects as object-oriented message ports: calling the `set` method sends a message to all replicas informing them of the new value of the data item, with the callback objects being used to receive these messages.

9.2 Multi-person Spaceframe Construction

In this section we present a simple example of how Repo's general purpose data sharing satisfies our goal of supporting exploratory programming of distributed augmented environments. As part of the Augmented Reality for Construction (ARC) project [9], we built an AR system to assist with the construction of space frame buildings. Our system prompted the worker by displaying the next part to be installed in the correct location on the partially completed space frame, as shown in Figure 3(a). After this prototype was working, we wished to explore how an AR construction assistant could be leveraged in other ways, such as allowing workers to discuss problems with a remote expert. The first step in this exploration was to create a visualization of the construction site, showing the status of the space frame being constructed, the location of the worker and the next piece to be installed, as shown in Figure 3(b).

The new visualization prototype first needed to share the state of the construction task with the ARC prototype. Therefore, we modified the ARC prototype to move its single state variable (`step`, representing the current task step) into a replicated object, and exported this variable to the network. We imported this variable into our remote monitoring prototype, and allowed both programs to change the construction step. However, we noticed that this did not give us all the information a remote monitor would need, especially information about when the worker performed incorrect actions. To distribute this information, we added routines to the replicated object that are called when various interesting conditions are noticed, shown in Figure 4(a). These conditions include the task being completed (`done`), the user scanning the wrong part (`wrongPart`) and the user scanning the correct part in the wrong location (`wrongPosition`). Notice that the `wrongPart` and `wrongPosition` methods do nothing; they are simply used to distribute a message, which can be noticed and reacted to in a callback notification object (as is done in Figure 4(b)).

This example illustrates the simplicity of prototyping with Repo. Modifying the code to access the construction step variable from the replicated object was trivial, as was adding the `wrongPart` and `wrongPosition` notification methods. In addition, this information is "typical" application data, and Repo allows us to distribute it to, and react to changes in, the various programs with a few lines of code that took a few minutes to write.

```
let stepObj = {replicated,
       step => initialStepNumber,
       done =>
          umeth (s)
            s.step := -1
          end,
       wrongPart =>
          umeth (s, barcode)
          end,
       wrongPosition =>
          umeth (s)
          end
    };
```

(a) The replicated state variable.

```
let stepCB = {simple,
     post‘step => meth(s,o,v)
          goToStep(v);
        end,
     post‘done => meth(s,o)
          goToStep(-1);
        end,
     post‘wrongPart => meth(s,o,v)
          doWrongPart(v);
        end,
     post‘wrongPosition => meth(s,o)
          doWrongPos();
        end
  };

let stepNotify =
  replica_notify(stepObj, stepCB);
```

(b) The notifier for the replicated state variable.

Figure 4: The replicated state for the distributed ARC prototype. By moving the state of the ARC prototype into a replicated variable, we can share it between multiple processes. Each process can create a notifier variable, similar to the one shown in (b) to perform whatever action is desired to react to the change. In this example, taken from the prototype code, the notification methods contain calls to other procedures defined elsewhere in the code, but could contain arbitrary code.

9.3 Hierarchical Object Directories

In this section, we will describe how hierarchical *object directories* (HOD) would be implemented in Repo. The HOD is a good example of how the three object semantics can be straightforwardly combined to form a complex and interesting data structure. This example also shows how nested references “do the right thing” when the object containing them is copied. In this case, client-server and synchronized replicated objects are contained in an unsynchronized replicated object. When the unsynchronized object is copied between sites, the embedded references to the other objects are copied appropriately, as a programmer would expect.

The goal of the object directories is to provide a mechanism for structuring applications that is useful for both stand-alone and distributed applications. The HOD provides object lookup and management, and is similar in flavor to directories in a file system, hierarchical environments used in many virtual environment systems, or name spaces used in many different application domains. We will not include the code for our HOD, as it is fairly long. It was designed to be analogous to a file-system, with a single object directory (OD) containing a set of key-value pairs that associate objects with textual names. References to virtually any kind of object can be stored in an OD.

In addition to allowing us to meaningfully assign names to services and resolve those names to computer addresses, the HOD can serve as the primary structuring metaphor for a family of distributed applications. By allowing an OD to contain references to other ODs, we can organize the HODs into a single global name space that allows applications to communicate with each other in a meaningful way. Within this hierarchy, data can be organized in well-defined subhierarchies so that applications know where to look for particular kinds of data and services. Furthermore, by allowing clients to watch one or more ODs for changes, such as the addition or deletion of entries, clients can react to changes in the world without the need for direct communication with the instigator of those changes.

To build a simple OD, each of the three types of objects are used, as shown in Figure 5. The OD itself is implemented using an unsynchronized replicated *wrapper* object (Figure 5(a) and (d)). This object has data fields and methods to implement the OD functionality. For example, there are methods to add elements to, or delete elements from, the OD. In addition to any other incidental data, the OD contains references to two important objects in its data fields, a *storage directory* and a *notifier directory.* The *storage directory* is a client-server object that imple-

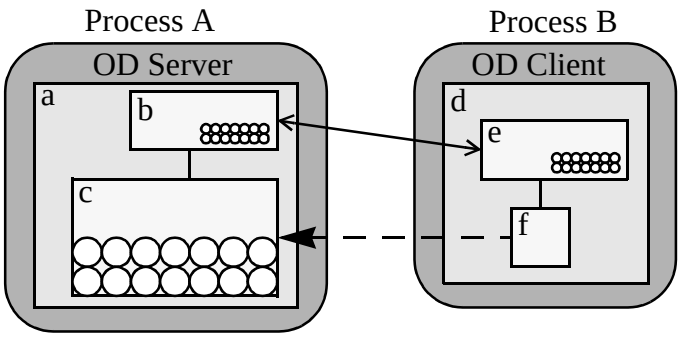


Figure 5: A single Object Directory (OD). The server on the left consists of (a) an unsynchronized replicated object acting as a wrapper, (b) a synchronized, replicated object implementing the notifier directory and (c) a client-server object implementing the storage directory. The client on the right has (d) its own copy of the wrapper object, (e) a shared copy of the notifier directory, and (f) a remote reference to the storage directory.

ments a centralized object store using key-value pairs (Figure 5(c)). It contains the actual data objects stored in the OD. The *notifier directory* is a synchronized replicated object that contains a small, constant-size piece of information for each entry in the directory, such as the type of the object, and is also used to receive notification of changes to the directory (Figure 5(b) and (e)).

Because the storage directory is implemented with a client-server object, its contents are not replicated. The single copy is accessed via remote method calls from any process that receives a copy of the OD. Conversely, since the notifier directory is implemented with a synchronized replicated object, it is fully replicated in all processes that receive a copy of the OD, with any updates to the OD distributed to it. There are three things to understand about why the OD is designed this way: what happens when an OD is passed to a remote process, how OD methods are implemented, and how Callback Objects are used by the OD.

First, consider what happens when an OD is passed from one process to another, as a parameter or return value of a client-server or replicated object method call. Since the OD is an unsynchronized replicated object, a new, independent copy of the object is created in the second process. As part of the process of creating that copy, the data fields of the object are copied using their semantics. Therefore, the new process will contain a new replica of the notifier directory, and a remote reference to the storage directory (Figure 5(f)). All of this happens automatically when a reference to the OD is transmitted to a new process.

Given the structure shown in Figure 5, how are OD methods implemented? Consider adding an element to an OD with a simple *put* method, "*put(name, object)*," which stores *object* in the OD under the key *name*. The *put* method of the OD would perform the following actions:

- store the object in the storage directory using the corresponding storage directory *put* method, and
- store the type of the object in the notifier directory using the corresponding notifier directory *put* method, which has been designated as an update method.

No matter which process performs the *put* operation, the outcome is the same:

- a remote procedure call is performed to store the object in the central storage directory, and
- the type of the object is stored in the shared notifier directory in all replicas, causing any Callback Objects registered for these replicas to have their *put* notifier methods invoked.

The wrapper object would use the Callback Objects associated with the notifier directory to monitor an object directory for changes on behalf of local clients that have requested notification when the OD changes.

10 Discussion and Future Work

We have used Repo to build a number of prototypes (e.g., [7]), and our experiences have been mostly positive. The ability to quickly and effortlessly create distributed applications using arbitrary combinations of objects, arrays and variables with both client-server and replicated distribution semantics has allowed us to concentrate on the applica-

tions and the interaction techniques we are interested in exploring, and take the distribution of data largely for granted.

While programmers can also build data structures in Modula-3 that combine these distribution semantics, courtesy of the Shared and Network Objects packages, exploratory programming in an interpreted language such as Repo is significantly faster. Furthermore, Repo's dynamic type system, and the ability to distribute arrays and variables in addition to objects, gives the programmer greater flexibility. While Shared and Network Objects can be used just as normal objects, Modula-3's strong static typing combined with the requirement that these objects inherit from different distinguished types means that a programmer can not mix them quite so freely as objects can be mixed in Repo. For example, in Modula-3, a procedure must be defined to take one of a Network or Shared Object as a parameter, but in Repo any data value can be passed as a parameter to the same procedure (it is up to the programmer to ensure that correct values are used in the correct locations).

However, there is a price to be paid for the increased flexibility of a dynamically typed language, and that is the greater difficulty in tracking down bugs; since procedures are untyped, incorrect usage may not cause errors immediately, since the variables themselves may not be used immediately. While the prototypes we have been creating have typically only been a few hundred to a few thousand lines of code, we have experienced problems debugging some of the larger ones. It was these problems that motivated us to create the reflection module (see [21]) to allow programmers to add type checking and controlled object access to their programs when they see fit. By judiciously checking parameters in a few key locations in a program, debugging of programs has been greatly simplified.

We have also learned some lessons about our design of Repo. One relates to the usefulness of the custom pickling facilities. It turns out that programmers need to be told that the pickling facilities in Repo are much less efficient than in Modula-3, and should not be used to try and obtain small performance improvements. In Modula-3, and other compiled languages, picklers are associated with objects of a certain type, and are compiled into all instances of the program. Therefore, aside from sending a small value to identify the type, only the data generated by the pickling routine is sent across the network. In Repo, on the other hand, there are no object types, so these pickling objects are associated with *instances* of Repo objects. Therefore, before these pickling object methods are run, the infrastructure must copy the basic object structure, including the pickling objects themselves and all the object methods, between processes. As a result, attaching custom pickling objects to an object initially increases the amount of information that is sent over the network, and is therefore useful primarily for situations where correctness, rather than efficiency, is the motivation for creating the custom pickler. For example, condition variables can not be copied over the network, so if an object contains one, the programmer needs to use a custom pickler to create a new one at the destination site.

Finally, we have noticed that novice programmers tend to forget to specify the distribution semantics of data values as they are programming, either because they are not thinking about distribution, or because they are implicitly assuming that the default is unsynchronized replication (the semantic most like data in a traditional programming language). We suspect that if we required all data values (arrays, objects and variables) to have their distribution semantics specified, instead of having objects default to client-server sharing with replication as an option, novices would learn to think about the semantics of the objects they are creating more quickly, and experienced programmers would not introduce bugs into their programs by forgetting to specify the semantics. However, this problem, and the others mentioned in this section, are relatively minor, especially in relation to Repo's advantages for building distributed applications.

Based on these experiences, there are a number of directions we would like to go in the future. Repo is well suited to exploratory programming of tightly-coupled, distributed, highly interactive systems. Our choice of the distributed object memory (DOM) programming model, and the approach we took to providing replicated data within that model, were guided by both the application domain and the exploratory style of programming in which we engage. In the future, we hope both to continue building on this approach to prototyping distributed interactive applications, and to explore different programming models that may be more appropriate to different domains and programming styles.

This latter question is an important one. While tightly-coupled, strictly consistent objects that are distributed using a DOM programming style are useful for exploratory programming, they may not be the most appropriate choice for other domains. For example, if one is building long lived, production quality systems, the trade-offs made between ease-of-use and efficiency might be different; efficiency of execution and network utilization are

likely to be much more important than the ease of changing objects from one distribution semantic to another, not to mention the increased importance of other issues such as fault tolerance. Therefore, the transparency with which the objects are integrated into the programming languages may not be the most important issue, as it is for us.

However, even with the programming style with which we are familiar, there are a number of ways we envision improving our implementation of Repo's DOM programming model: by decreasing the latency of update distribution, improving network awareness so programmers could find out more about the distribution patterns of their objects, adding additional per-object replication semantics as the need arises and extending the programming model to support multi-object operations. Finally, we would like to explore these ideas in other programming languages, especially Java.

11 Acknowledgments

I would like to thank the members of the Computer Graphics and User Interfaces Lab at Columbia University, where this work was undertaken, especially the lab director Steven Feiner. Numerous others at Columbia influenced this work over the years, especially Clifford Beshers, Sushil Dasilva, Tobias Höllerer and Steven Dossick. Luca Cardelli of DEC (now Compaq) SRC created Obliq, and provided ongoing help over the years that Repo was developed. Bill Kalsow and Farshad Nayari of Critical Mass provided help with all aspects of Modula-3, including creating an excellent implementation of the language.

This work was supported by ONR Contracts N00014-94-1-0564 and N00014-97-1-0838, the National Science Foundation under Grant CDA-92-23009 and ECD-88-11111, the New York State Center for High Performance Computing and Communications in Healthcare (supported by the New York State Science and Technology Foundation), the Advanced Network & Services National Tele-Immersion Initiative, and gifts from NYNEX Science & Technology, Critical Mass and Microsoft.

12 References

1 Bal, H., Kaashoek, M., and Tanenbaum, A. (1992). Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205.

2 Bal, H. E. and Tanenbaum, A. S. (1988). Distributed programming with shared data. In *Proc. of the 1988 Int'l Conf. on Computer Languages*, pages 82–91.

3 Birrell, A. and Nelson, B. (1984). Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2(1):39–59.

4 Birrell, A., Nelson, G., Owicki, S., and Wobber, E. (1993). Network objects. In *Proc. 14th ACM Symp. on Operating Systems Principles*.

5 Cardelli, L. (1995). A language with distributed scope. *Computing Systems*, 8(1):27–59.

6 Decouchant, D. (1986). Design of a distributed object manager for the Smalltalk-80 system. *ACM SIGPLAN Notices*, 21(11):444–444.

7 Feiner, S., MacIntyre, B., Höllerer, T., and Webster, A. (1997). A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. *Personal Technologies*, 1(4):208–217.

8 Feiner, S., MacIntyre, B., and Seligmann, D. (1993b). Knowledge-based augmented reality. *Communications of the ACM*, 36(7):52–63.

9 Feiner, S., Webster, A., Krueger, T., MacIntyre, B., and Keller, E. (1995). Architectural anatomy. *Presence: Teleoperators and Virtual Environments*, 4(3):318–325.

10 Gray, R. S. (1996). Agent Tcl: A flexible and secure mobile-agent system. In *4th Annual Tcl/Tk Workshop '96*, pages 9–23, Monterey, CA.

11 Grimshaw, A. S. (1993) Easy-to-Use Object-Oriented Parallel Processing with Mentat, *Computer*, 26(5): 39–51.

12 Harbison, S. P. (1992). *Modula-3*. Prentice-Hall.

13 Haridi, S., Van Roy, P., and Smolka, G. (1997) An Overview of the Design of Distributed Oz, In *Proceedings*

of the Second International Symposium on Parallel Symbolic Computation (PASCO '97), pages 176–187.

14 Janssen, B., Spreitzer, M., Larner, D., Jacobi, C. (1998). ILU Reference Manual. Xerox Palo Alto Research Center, Palo Alto, CA.

15 Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-grained mobility in the Emerald system. *ACM Trans. on Computer Systems*, 6(1):109–133.

16 Kristensen, A. and Low, C. (1995). Problem-oriented object memory: Customizing consistency. In *Proc. ACM OOPSLA '95*, pages 399–413.

17 Levelt, W., Kaashoek, M., Bal, H., and Tanenbaum, A. (1992). A comparison of two paradigms for distributed shared memory. *Software Practice and Experience*, 22(11):985–1010.

18 Li, K. (1986). *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Department of Computer Science, Yale University.

19 Liskov, B. (1988). Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312.

20 MacIntyre, B. (1997). COTERIE: Columbia object-oriented toolkit for exploratory research in interactive environments. In *IEEE WETICE '97 Workshop on Distributed Systems Aspects of Sharing a Virtual Reality*, Cambridge, MA.

21 MacIntyre, B. (1999). Exploratory Programming of Distributed Augmented Environments. PhD thesis, Department of Computer Science, Columbia University.

22 MacIntyre, B. and Feiner, S. (1996a). Future multimedia user interfaces. *Multimedia Systems*, 4(5):250–268.

23 MacIntyre, B. and Feiner, S. (1996b). Language-level support for exploratory programming of distributed virtual environments. In *Proc. UIST '96*, pages 83–94, Seattle, WA.

24 MacIntyre, B. and Feiner, S. (1998). A distributed 3D graphics library. In *Computer Graphics (Proc. ACM SIGGRAPH '98)*, Annual Conference Series, pages 361–370, Orlando, FL.

25 Nog, S., Chawla, S., and Kotz, D. (1996). An RPC Mechanism for Transportable Agents. Technical Report PCS-TR96-280, Dartmouth College, Computer Science, Hanover, NH.

26 OMG (1992). *The Common Object Request Broker: Architecture and Specification*. Object Management Group, Inc., Framingham, MA, 1.1 edition.

27 Ousterhout, J. K. (1990). Tcl: An embeddable command language. In *USENIX Conference Proceedings*, pages 133–146.

28 Perham, M., Smith, B. C., Janosi, T., and Lam, I. K. (1997). Redesigning Tcl-DP. In *5th Annual Tcl/Tk Workshop '97*, pages 49–53, Boston, MA.

29 van Rossum, G. (1995). Python library reference. Technical Report CS-R9524, CWI - Centrum voor Wiskunde en Informatica.

30 Weiser, M. (1991) The Computer for the 21st Century. *Scientific American*, 265(3):94–104.

31 Wilson, G. and Bal, H. (1996) Using the Cowichan Problems to Assess the Usability of Orca. *IEEE Parallel and Distributed Technology: Systems and Applications*, 4(3): 36–44.

32 Wilson, G. and Lu, P. (1996) Parallel Programming Using C++, The MIT Press, Cambridge, MA, USA.

33 White, J. E. (1994). Telescript technology: The foundation for the electronic marketplace. White paper, General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040.

34 Wollrath, A., Riggs, R., and Waldo, J. (1996). A distributed object model for the Java system. *Computing Systems*, 9(4):265–290.